Characterization and Modeling of PIDX Parallel I/O for Performance Optimization

Sidharth Kumar*, Avishek Saha*, Venkatram Vishwanath†, Philip Carns†,
John A. Schmidt*, Giorgio Scorzelli*, Hemanth Kolla‡, Ray Grout§, Robert Latham†,
Robert Ross†, Michael E. Papka†±, Jacqueline Chen‡, Valerio Pascucci*×

*Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, UT, USA
†Argonne National Laboratory, Argonne, IL, USA
×Pacific Northwest National Laboratory, Richland, WA, USA
‡Sandia National Laboratories, Livermore, CA, USA
±Northern Illinois University, DeKalb, IL, USA
§National Renewable Energy Laboratory, Golden, CO, USA
sidharth@sci.utah.edu

ABSTRACT

Parallel I/O library performance can vary greatly in response to user-tunable parameter values such as aggregator count, file count, and aggregation strategy. Unfortunately, manual selection of these values is time consuming and dependent on characteristics of the target machine, the underlying file system, and the dataset itself. Some characteristics, such as the amount of memory per core, can also impose hard constraints on the range of viable parameter values. In this work we address these problems by using machine learning techniques to model the performance of the PIDX parallel I/O library and select appropriate tunable parameter values. We characterize both the network and I/O phases of PIDX on a Cray XE6 as well as an IBM Blue Gene/P system. We use the results of this study to develop a machine learning model for parameter space exploration and performance prediction.

Keywords

I/O & Network Characterization, Performance Modeling

1. INTRODUCTION

PIDX, a high-performance parallel I/O library, writes data in a multiresolution, Hierarchical-Z (HZ) order layout [29]. This enables interactive visualization of large-scale simulations, and is of critical importance in meeting the increasing gap between the compute and storage capabilities of supercomputers. PIDX uses a flexible, customized collective I/O algorithm [8] that expands on two-phase I/O techniques [19] to support parallel HZ encoding and maintains compatibility with the existing IDX data format. Collective I/O in PIDX involves three phases: restructuring, aggregation, and file system writes. Each of these phases offers user-tunable parameters that substantially influence I/O performance.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SC'13, November 17-22, 2013, Denver, Colorado USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11 . . . $15.00
http://dx.doi.org/10.1145/2503210.2503252

The performance of parallel I/O libraries such as PIDX is influenced by characteristics of the target machine, characteristics of the data being accessed, and the value of tunable algorithmic parameters. HPC systems vary widely in terms of network topology, memory, and processors. They may also use different file systems and storage hardware that behave differently in response to I/O tuning parameters. Although many of these properties can be empirically determined through characterization studies, it is still difficult to translate them into an optimal set of tuning parameters for a sophisticated I/O library. In this situation it is helpful to construct a model of the system to aid in exploration of the parameter space. Moreover, the model can be used to suggest promising parameter configurations and autotune the system for higher levels of performance.

In this work, we present a characterization study of the PIDX collective I/O algorithm including both network aggregation and file system I/O. Our goal is to understand how performance is affected by combinations of fixed input parameters (system and data characteristics) and tunable parameters. We then model the behavior of the PIDX parallel I/O library using machine learning techniques. Earlier work in high-performance research has proposed analytical models, heuristic models, and trial-and-error based approaches. All these methods have known limitations [9] and do not generalize well to a wide variety of settings. Modeling techniques based on machine learning overcome these system limitations and build a knowledge-based model that is independent of the specific hardware, underlying file system, or custom library used. Based on the flexibility and independence to a variety of constraints, machine learning techniques have achieved tremendous success in extracting complex relationships just from the training data itself.

In this paper, we build models using regression analysis on data sets collected during the characterization study. The regression models predict PIDX performance and identify optimal tuning parameters for a given scenario. Our models

have been trained on data obtained from experiments conducted over a small number of cores. We first validate our models on (training) datasets from low core count. Then we use these models (and a small number of samples from simulations on high core count) for throughput prediction on test datasets from the high core count regime. Our goal is to show that such a model would be useful for *approximately* predicting the behavior of a system in higher core count scenarios where brute-force sensitivity studies would be both costly and resource intensive. The samples from the high core count regime are obtained by augmenting the regression model with a sampling technique. Consequently, we obtain a new model that adaptively improves itself over multiple simulation time steps, thus autotuning the system parameters to achieve higher performance over time. We use PIDX as our case study for characterization and modeling in this work and, in Section 5.3, discuss how our techniques can be applied to other I/O software stacks. We demonstrate the impact of the adaptive PIDX modeling techniques on the S3D combustion application.

The main contributions of our paper are the following: (a) using PIDX as a benchmark to develop a methodology to evaluate characteristics of the storage system and network that contribute to its performance, (b) characterizing two different architectures (Hopper and Intrepid) as case studies to show that these characteristics can vary significantly across platforms, (c) investigating models with the goal of predicting the PIDX I/O library performance, and (d) demonstrating the application of those models to the PIDX library, although a similar methodology should be applicable to other applications or I/O libraries as well. In addition, we report the following key findings from our characterization and modeling study. First, we found that Hopper network is more sensitive than is Intrepid to variations in the quantity and size of network messages. Second, owing to differences in ways job partitions are allocated, the two architectures exhibited different network scaling behaviors. Third, Hopper (Lustre) is more optimized to a unique file per process I/O approach than is Intrepid (GPFS), whose I/O is optimized for fewer shared files. Moreover, for varying data load Hopper showed variation in performance pattern with similar parameter configuration. In the modeling study, we observed small validation errors for throughput prediction on a low number of cores and comparatively higher error on high core count experiments. Overall, as we discuss in Section 5.3, we found Hopper was more difficult to model (particularly at high core counts) compared with Intrepid.

The paper is organized as follows. In Section 2 we present background information on PIDX with an emphasis on its parameters. In Section 3 we present our experimental platforms. In Section 4 we present the results of our characterization study (network and I/O) and explain the methodology adopted as well the results obtained for shuffling, aggregation, and combined phases. In Section 5 we present our model and prediction results for performance throughput and tunable parameters. We show results on microbenchmarking as well as S3D combustion data. We discuss relevant literature in Section 6 and end the paper with our conclusions in Section 7.

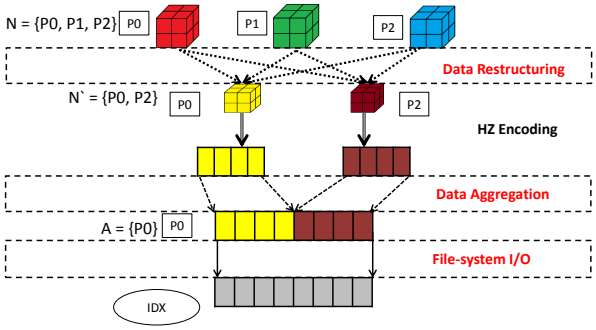


Figure 1: Block diagram showing the three I/O phases of PIDX – restructuring, aggregation, and file I/O.

2. PARALLEL IDX (PIDX)

IDX is a file format suitable for visualization of large scale HPC simulation results due to its ability to access data at multiple levels of resolution with low latency for interactive exploration [30]. IDX uses hierarchical Z (HZ) ordering to reorder data (at multiple resolutions) based on its spatial location in each dimension. This hierarchical, multiresolution data representation is beneficial for large-scale interactive analysis and visualization [39]. However, the file format, while successful in the areas of digital photography [34] and large scale visualization [29], is inherently serial. In earlier work [17] we proposed a three-phase I/O approach to write data in parallel in IDX file format. The first phase involved restructuring of simulation data into large blocks (powers of 2) while preserving the original multidimensional format. This facilitated optimized HZ ordering followed by efficient I/O aggregation (second phase) and ending with actual disk-level I/O writes (third phase). By adopting this three-phase I/O we were able to mitigate the shortcomings of small disk accesses as well as unaligned and discontinuous memory access. Figure 1 illustrates the three phases of PIDX.

Three Phases of PIDX. In the first I/O phase (data restructuring [18]), data is redistributed from initial N processes to N' processes in larger power-of-2 sized blocks. PIDX allows two values of N': (1) default, N (or $\sim N$), where the extents of the restructured block have dimensions rounded to the closest power-2 number (of the per-process volume), and (2) expanded, $N/8$ (or $\sim N/8$), where the dimension (or length) of the restructured block is twice that of the default case. In the second I/O phase (aggregation in Figure 1), data after restructuring is aggregated from N' processes to a chosen set of A aggregators. The PIDX aggregation phase can be tuned in two ways: by varying the number of processes participating in aggregation and by varying the number of aggregators. The first is a consequence of the restructuring phase that controls the number of processes participating in aggregation ($N' \sim N$ or $N/8$). The second can be directly tuned with PIDX. The default number of aggregators in PIDX is a product of tunable parameter F (file count) and input parameter V (variable count). However, this scheme which corresponds to the case where an aggregator is responsible for writing all data for one variable in one single file lacks flexibility. To overcome this lack of flexibility, we introduce an additional parameter a_f (aggregation

Independent Variables (Input)	Parameters (Tunable)	Dependent Variables (Output)
N: Initial set of nodes	N': Set of nodes after re-structure[1]	Network throughput
D: Data size/resolution	A: Number of aggregators = $(a_f \times V \times F)$[2,3]	I/O throughput/File
V: Number of variables	a_f: aggregation factor [2,3]	Combined throughput
M: Machine specs/memory	F: File counts[3]	

Table 1: Summary of parameter space. Numerical superscript refers to the I/O phase.

factor). Instead of having a fixed number of aggregators as in the default case, a_f provides the flexibility to use all available cores as aggregators. So now the aggregator count A is expressed as $a_f \times F \times V$, such that $A \leq N$. The third I/O phase (disk-level write) is directly influenced by both aggregator count (A) and file count (F).

PIDX Parameters. Table 1 summarizes the parameter space for PIDX. The table is divided into input, output, and tunable parameters. Input parameters are preset at the start of the simulation design. Tunable parameters can be adjusted to improve and optimize performance over multiple simulation runs. In the tunable parameter column the numerical superscript refers to the I/O phase. For example, N' belongs to restructuring (the first I/O phase).

In the dependent output parameter column we have network throughput, I/O throughput, and the combined throughput of network and I/O. Since PIDX has its own customized collective I/O implementation involving both data aggregation and disk-level I/O phases, we examined both phases separately as well as together.

3. EXPERIMENTAL PLATFORMS

The experiments presented in this work were performed on Hopper at the National Energy Research Scientific Computing (NERSC) Center and Intrepid at the Argonne Leadership Computing Facility (ALCF). Hopper is a Cray XE6 with a peak performance of 1.28 petaflops, 153, 216 compute cores, 212 TiB of RAM, and 2 PiB of online disk storage. All experiments on Hopper were carried out using a Lustre scratch file system composed of 26 I/O servers, each of which provided access to 6 Object Storage Targets (OSTs). Unless otherwise noted, we used the default Lustre parameters that striped each file across two OSTs. Intrepid is a Blue Gene/P system with a peak performance of 557 teraflops, 167, 936 compute cores, 80 TiB of RAM, and 7.6 PiB of online disk storage. All experiments on Intrepid were carried out by using a GPFS file system composed of 128 file servers and 16 DDN 9900 storage devices. Intrepid also uses 640 I/O nodes to forward I/O operations between compute cores and the file system. The Intrepid file system was nearly full (95% capacity) during our evaluation study. We believe that this situation significantly degraded I/O performance on Intrepid.

4. CHARACTERIZATION METHODS

The most critical tunable parameter for both the network and I/O phases of PIDX is the selection of the number of aggregator processes (A). This parameter impacts the overall algorithm in three ways: (1) it affects the distribution of data over the network when aggregating data from the N' cores that hold restructured data, (2) it dictates the number of cores that will access the file system during the I/O phase, and (3) it controls the maximum amount of memory that will be consumed on a core. The memory consumption is particularly important for applications that have already been tuned to use a large fraction of memory on each core for computation purposes. In this work we assume that all data will be transferred to the aggregators before being written to the file system. In future, we also plan to explore pipelining techniques to limit the aggregator memory consumption.

We varied the number of aggregators A in our study from N' (where every process holding restructured data is also an aggregator) to $N'/32$ (where one of 32 processes holding restructured data is also an aggregator). The $A = N'$ case uses the least amount of memory per processes, whereas $A = N'/32$ requires the most memory.

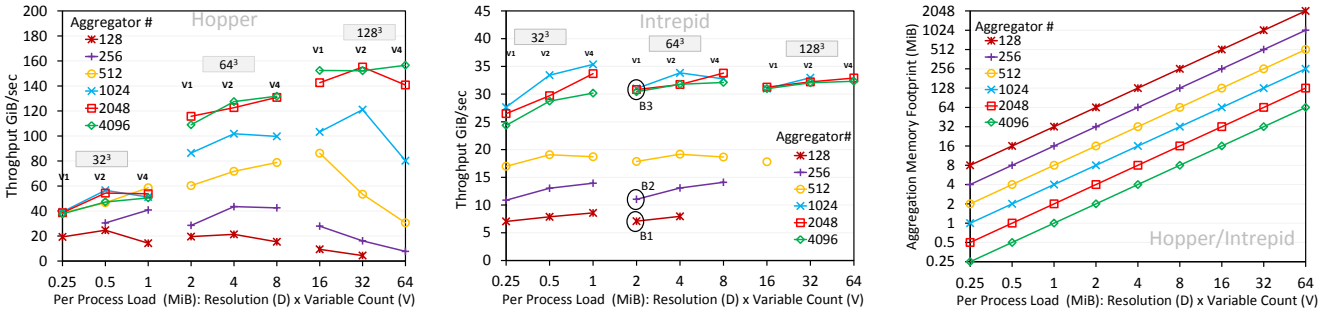
We have some flexibility in varying the value of N'. This affects two aspects of the network phases: (1) how data is redistributed from the compute cores (N) to restructuring cores (N') for HZ encoding during the restructuring phase and (2) how many nodes the data must be aggregated from during the aggregation phases. The default value of N' is N, but we also evaluate an *expanded* restructuring configuration in which $N' = N/8$. The value of N' has no effect on the I/O phase of the algorithm.

We used two types of experiments to characterize network and I/O performance. In the first type, which we refer to as *data scaling*, the number of cores is fixed at 4096 with all cores participating in aggregation (*default* case) while the per-process data size is exponentially varied from 256 KiB to 64 MiB. Table 2 lists the values, for resolution (D) and variable count (V), used to achieve the range of data size within each process. In the second type, which is *weak scaling*, we fix the per process data resolution to $64 \times 64 \times 64$ ($V = 1, 2, 4$) and vary the number of cores between 1024, 4096 and 8192.

4.1 Network characterization

Resolution (D)	Variables (V)	Memory
$32 \times 32 \times 32$	1	256 KiB
	2	512 KiB
	4	1 MiB
$64 \times 64 \times 64$	1	2 MiB
	2	4 MiB
	4	8 MiB
$128 \times 128 \times 128$	1	16 MiB
	2	32 MiB
	4	64 MiB

Table 2: Different configuration of resolution and variables of dataset used in our experiment for data scaling along with corresponding memory allocated.



(a) (b) (c)

Figure 2: Throughput of data shuffling phase with varying data loads for (a)HOPPER and (b)INTREPID and (c)Memory footprint for all aggregator counts.

In this section we study the performance of data aggregation from N' processes to A aggregators to characterize the interconnect network. One-sided remote memory access (RMA) is used for all communication in this phase of the PIDX algorithm.

Data Scaling. Network scaling results for HOPPER and INTREPID can be seen in Figure 2. Note that the figure has disconnected trend lines, since we group them by resolution values (annotated by a gray box at the top of each column). Overall, an increase in aggregator count typically leads to an improvement in performance on both machines and across all data loads (32^3, 64^3 and 128^3 with $V1, V2, V4$).

We collected network counter data for INTREPID to better understand the impact of different aggregation schemes. The data corresponds to all links of the torus for the three aggregator counts (4096, 256, and 128) and for data size 64^3 with one variable (see Figure 3). We use a projection of the 3D network topology provided by Boxfish [21], an integrated performance analysis and visualization tool developed at Lawrence Livermore National Laboratory. Each image of Figure 3 shows all the network links along two torus dimensions aggregated into bundles along the third dimension. Two observations can be made from the figures: (1) fewer data packets are transmitted across the network with increasing aggregators, and (2) reduced aggregator count results in skewed data distribution across network links. Hence, for INTREPID the higher aggregator counts of N', $N'/2$, and $N'/4$ perform much better than the lower aggregator counts $N'/8$, $N'/16$, and $N'/32$. Also, for INTREPID almost similar performance for N', $N'/2$, and $N'/4$ can be attributed to the architecture of the machine, where it has 4 cores on a node. Boxfish visualization data is not available on HOPPER, but we believe that our performance analysis for INTREPID holds for HOPPER as well, since they demonstrate similar performance ordering with varying aggregator counts (see Figures 2a and 2b).

We also observe that the range of aggregator counts from 128 to 1,024 exhibit a degrading trend on HOPPER as the data volume is increased. In contrast, INTREPID exhibits fewer variations across data volumes regardless of the number of aggregators. This result can be attributed to the fact that

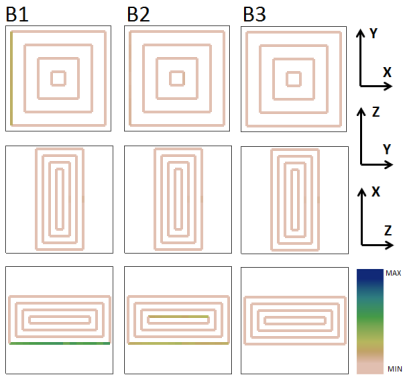


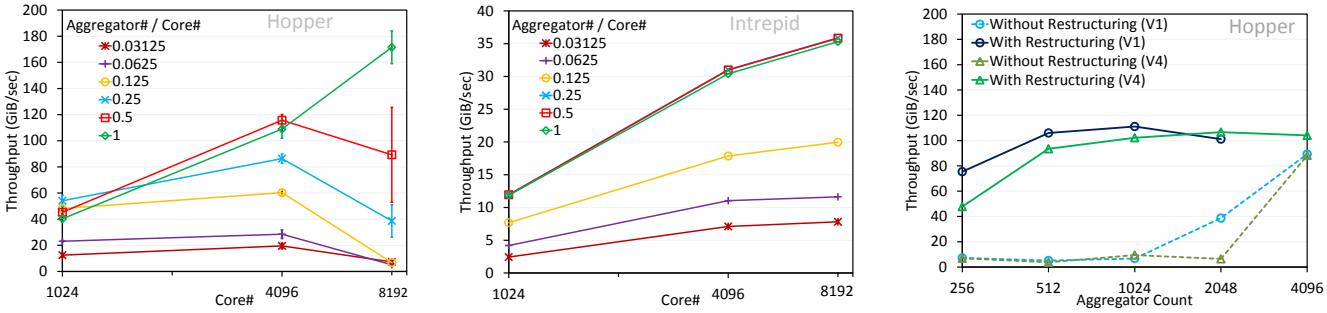
Figure 3: Visualizing the INTREPID network flow with Boxfish for aggregator count 128 (B1), 256 (B2), and 4,096 (B3) (denoted as black circles in Figure 2b).

HOPPER is more sensitive in variations in the quantity and size of network messages being transmitted between nodes.

Figure 2c illustrates how memory usage varies inversely with the number of aggregators on either system. A non-uniform distribution of data requires some processes to allocate more memory than others, thus increasing the total memory usage for smaller aggregator counts.

Weak Scaling. Figure 4 shows network scalability on HOPPER and INTREPID while varying the total number of cores from 1,024 to 8,192 with a fixed data volume per process of 64^3. We plot trendlines corresponding to the ratio of number of aggregators and the number of cores. We use six ratios of $1/32 = 0.03125$, $1/16 = 0.0625$, $1/8 = 0.125$, $1/4 = 0.25$, $1/2 = 0.5$, and 1, where a ratio of 1 implies aggregator count equal to the total number of cores.

In Figure 4a, HOPPER shows scalability with 8K cores only when the number of aggregators and processes participating in aggregation are equal. In contrast, INTREPID shows an upward trend for all core counts in Figure 4b regardless of the aggregator ratio. These differences in behavior are a result of the different network topologies in HOPPER and INTREPID. INTREPID has a dedicated network partition for each application, while HOPPER has a shared network



(a) (b) (c)

Figure 4: Scaling of throughput performance for data-shuffling phase on (a) HOPPER, (b) INTREPID, and (c) throughput performance after restructuring data (before aggregation), and hence reducing the numbers of processes participating in aggregation to $8,192/8 = 1,024$.

and nodes may be allocated in a physically distant manner. With fewer aggregators and a larger number of processes participating in aggregation, a packet must travel a greater distance. This increases the likelihood of interference from other running jobs.

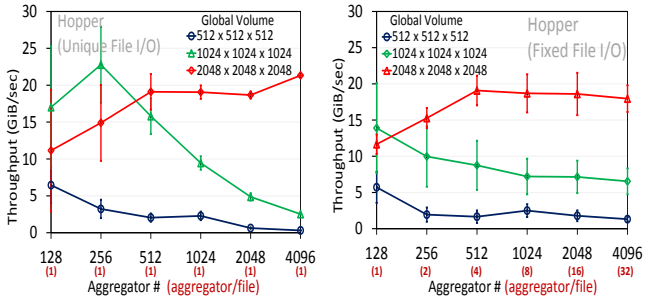
An additional restructuring phase led to the scaling of data-shuffling phase for HOPPER, as seen in Figure 4c. All the results are for $8,192$ cores while varying the number of aggregators from 256 to $4,096$. We show results for two variable count, 1 and 4 for data with resolution 64^3, thus varying the per-process load to 2 MiB and 8 MiB. Restructuring data using an *expanded* box before the aggregation phase reduced the number of processes participating in aggregation by one-eighth ($1,024$ from $8,192$) which indirectly reduced the outreach of nodes thereby compressing the networking space. As can be seen in the figure, at an aggregator count $4,096$, there is almost a tenfold improvement in performance for aggregation with restructuring over aggregation without restructuring.

4.2 I/O characterization

In this section we study the performance of file I/O as A aggregator nodes write to the parallel file system. As in the network study, we scale both load and core counts in order to evaluate different aspects of the storage system. We present excerpts from our study that exhibit interesting behavior.

Data Scaling. For HOPPER, with 4,096 cores, we used different global volumes of 512^3, $1{,}024^3$ and $2{,}048^3$. Each global volume setting consists of one variable with per process resolution of 32^3, 64^3, and 128^3 and varying data sizes of 256 KiB, 2 MiB and 16 MiB, respectively. We conducted two sets of experiments on HOPPER where, in each case, we either varied the total number of files while keeping the aggregation factor constant or varied the aggregation factor while keeping the number of files constant.

In the first set of experiments with both a_f and variable count equal to 1, the number of aggregators is equal to the number of files. We refer to this configuration as CASE U because of its similarity to a unique file per process I/O strategy. In the second set of experiments, we keep the number of files constant (set to the minimum file count seen in CASE U) and instead vary the number of aggregators using the variable aggregation factor a_f. We refer to this configuration as CASE F, since we keep the number of files constant. As in network characterization we exponentially vary the number of aggregators from N to $N/32$ ($4,096$ to 128).



(a) (b)

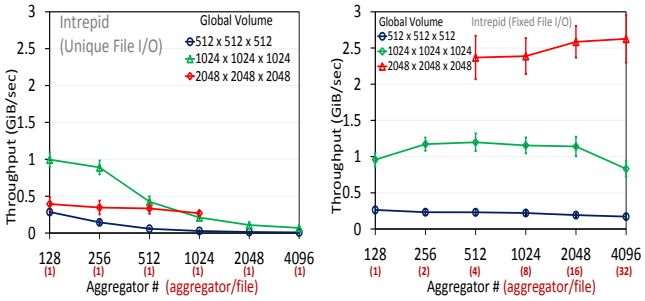
Figure 5: HOPPER: Throughput vs. aggregator count for (a) CASE U (unique File I/O) (b) CASE F (fixed File I/O).

Figure 5 shows the CASE U and CASE F results on HOPPER. CASE U exhibits superior peak performance compared with that of CASE F for all three data volumes, but the optimal number of aggregators varies in each case. If we focus on the 1024^3 volume as an example, we observe two key trends. First, the performance degrades as the aggregator count increases for both CASE U and CASE F. This result can be explained by looking at burst sizes for the different aggregator configuration in Table 3. A large number of aggregators leads to smaller I/O write sizes as the data is distributed over more processes. For example, for the 32^3 dataset with 128 aggregators, the I/O burst size is 8 MiB, whereas with 4,096 aggregators, it is only 256 KiB. The small write access pattern scales poorly. Second, CASE U exhibits more rapid degradation than does CASE F as the number of aggregators is increased. This difference is due to the overhead in creating a larger number of files in CASE U. For example, 4,096 unique files must be created when using 4,096 aggregators in CASE U. Although these files are created in parallel, the file creation cost constitutes a larger portion of the I/O time

relative to the cost of the actual file I/O. The 2048^3 global volume example does not exhibit this trend even though it is creating the same number of files because the I/O volume is large enough to amortize the file creation overhead. We therefore observe a steady improvement in performance as the aggregator count is increased at larger data volumes.

Agg Count	IO Burst Size (MiB) 32^3 (V1)	IO Burst Size (MiB) 64^3 (V1)	IO Burst Size (MiB) 128^3 (V1)
128	8	64	512
256	4	32	256
512	2	16	128
1024	1	8	64
2048	0.5	4	32
4096	0.25	2	16

Table 3: I/O Burst size for all aggregation combination with all different loads.



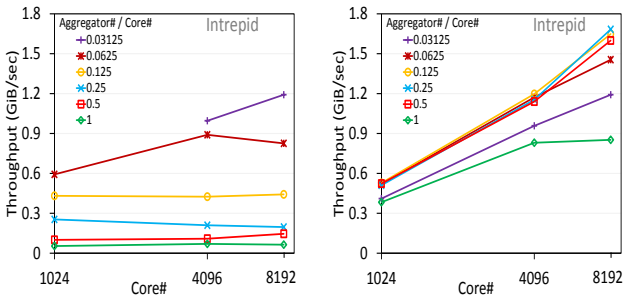
(a) (b)

Figure 6: Intrepid: Throughput vs. aggregator count for (a) Case U and (b) Case F.

We use the same experimental setup for measurements on Intrepid. The results are shown in Figure 6. Unlike Hopper, Intrepid shows better performance for Case F than for Case U for all data volumes. This is due to differences in file system architecture between the systems. We achieve better I/O performance on Intrepid by using fewer files (shared) and avoiding serialization points from creating multiple files in the same directory.

Weak Scaling. In this analysis we retain the Case F and Case U configurations from the *data scaling* analysis, but we vary the total number of cores while keeping the data volume per process fixed at 64^3 with one variable (2 MiB per process). We plot trendlines corresponding to the ratio of number of aggregators and the number of cores. We use six ratios of $1/32 = 0.03125$, $1/16 = 0.0625$, $1/8 = 0.125$, $1/4 = 0.25$, $1/2 = 0.5$, and 1, where a ratio of 1 implies aggregator count equal to the total number of cores.

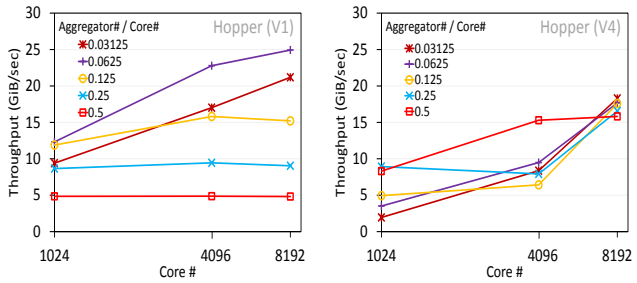
On Intrepid for both Case F and Case U, the weak scaling shown in Figure 7 reveals distinct performance patterns. As can be seen in Figure 7a, for Case U where the number of files equals the number of aggregators, we see two important trends: improvement in performance with decreasing aggregator count for all cores and no scaling in performance with increasing core counts. These observations are in direct accordance with the results we saw in the previous study (*data scaling*), with a decreasing number of aggregators performance increases. The trends are due to large I/O writes



(a) (b)

Figure 7: Throughput with increasing core counts; trend lines correspond to the ratio of aggregator count and core count for (a) Case U (b) Case F.

leading to better disk access patterns, as well as smaller overhead in creating the hierarchy of files. Straight trend lines across cores represent poor scaling performance, which can be explained by the increasing overhead in creating more files. In Figure 7b the performance scales according to the quantity and size of the I/O operations with no change in file creation overhead.



(a) (b)

Figure 8: Throughput with increasing core counts; trend lines correspond to the ratio of aggregator count and core count for (a) Case U (for V=1) (b) Case U (for V=4).

A similar weak scaling experiment on Hopper for Case U can be seen in Figure 8a. Performance behavior is similar to that of Intrepid with throughput improving with increasing aggregator count. The flat trendlines for the aggregator count and core count ratio 0.5 and 0.25 indicate no scaling due to the overhead in creating the hierarchy of files. A key point to note is the performance gain at every core while reducing the number of aggregators. Reducing the number of aggregators by half yields a twofold performance improvement (from 2.5 GiB/sec to 5 GiB/sec) due to both better disk-access pattern from large I/O writes and smaller overhead in creating the hierarchy of files. As can be seen from Table 3, for the 64^3 dataset with one variable and 4,096 aggregators, the I/O burst size is only 2 MiB as compared with relatively larger (and more favorable) burst sizes of 4 MiB and 8 MiB with aggregator counts 2,048 and 1,024. Besides better disk accesses with fewer aggregators, there are fewer files to write, and the overhead in creating the files is reduced as well. This observation is applicable to other libraries. File per process access patterns on this system are

most effective if the data volume is large enough to amortize file creation costs.

Instead of showing results for CASE F on HOPPER, we instead present a new set of experiments in which we change the number of variables to four while otherwise retaining the load configurations from Figure 8a and CASE U. This illustrates the performance of the system with a larger load. The results can be seen in Figure 8b. The key observation is the reversal of order of aggregator while going from 1024 cores to 8192 cores. Fewer aggregator counts along with fewer file counts results in better performance at higher scale because of the reduced contention and file creation overhead.

The above examples, specifically the last one, illustrate how dramatically system behavior can vary at different data scales. This situation motivates the need to build a performance model to aid in parameter exploration.

5. MODELING

In this section, we construct machine-learning-based system models using regression analysis. As we show, our learned models accurately predict the performance and parameters for a wide range of system configurations and for two different computing architectures, HOPPER's and INTREPID. Moreover, the models used are independent of PIDX and can be used for other existing parallel I/O libraries.

Model Description. To model a system, we collect training data, select informative feature or attributes from the training data, train different models, and choose the best-performing model. Apart from studying the system behavior, another benefit of the characterization study is generating lots of data which we can use to build a prediction model. Here each simulation run is a data point and the different parameter choices are the attributes (or attributes/fields/dimensions) of the data point. The total number of simulation runs defines the number of data points (or training data size, say N). Since the number of parameters is the same for each data point (say, D), we get a dataset of size N and attribute size D. Table 4 presents a typical example of a single data point with 9 attributes from our training dataset. One can think of the data point as a vector with D components. Let x_i denote the ith data point, and let x_{i1} to x_{iD} be its D attributes. In our example, x_{i1} would be "GlobalV," x_{i2} would be "LocalV," and so on. Each data point x_i is associated with an output y_i, which in our case is the throughput. Given a dataset $\{x_i, y_i\}$ of $i = 1, \ldots, N$ and $x_i \in \mathbb{R}^D$, a general machine learning model (call it θ) aims to learn the relationship between the D variables x_{i1} to x_{iD} and the observed outcome y_i. The learned model θ predicts the outcome $\hat{y}_i = \theta(x_i)$ for each data point x_i and in the process incurs a loss $\sum_{i=1}^{n} loss(\hat{y}_i, y_i)$. Loss quantifies the deviation of the predicted outcome from the true outcome and can be of various forms such as squared, absolute, log, or exponential. The goal of the learning process is to formulate an objective or cost function (that includes the loss) and then choose a model that best minimizes the objective (and hence the loss). The general form of the optimization problem to be minimized is

$$\arg\min_{\theta} \sum_{i=1}^{n} loss(\hat{y}_i, y_i) = \arg\min_{\theta} \sum_{i=1}^{n} loss(\theta(x_i), y_i),$$

and this can be solved by using standard techniques from the optimization literature (for example, gradient descent [4]). In algebraic terms, we can think of the dataset as an $N \times D$ matrix $\mathbf{X}$ and the outcome as an $N \times 1$ vector $\mathbf{Y}$; and the goal is to solve for the expression $\mathbf{Y} = \mathbf{X}\theta$ to obtain the $D \times 1$ model description vector θ.

Based on the problem domain or the data type, machine learning models can be broadly categorized into two types: (a) classification and (b) regression. In classification problems the outcomes y_i are categorical variables whereas for regression the output is continuous variable. In our study, because of the continuous nature of the throughput output variable, we build on regression based models. Regression models have the general form

$$y_i = \theta(x) = \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \ldots + \beta_D x_{iD} + \varepsilon_i,$$

where the model θ is represented by D coefficients β_j ($j = 1, \ldots, D$) and ε_i represents random noise in the data. As earlier, the goal is to solve for the regression coefficients β_js using algebraic analysis or optimization schemes.

Model Selection. Various regression models have been proposed in the literature. In this study, we experimented with a number of different regression models [6] that included (a) linear models, such as, Linear regression, Ridge regression, Lasso, Lars (Least angle regression), Elastic Net, SGD (Stochastic gradient Descent), Support Vector regression (with linear kernel), as well as, (b) non-linear models, such as, Decision trees, Support Vector regression (with polynomial and RBF kernels), Gaussian processes. Oftentimes, an *ensemble* of classifiers frequently outperform a single classifier. This has lead to the popularity of ensemble models, such as, Bagging and Boosting. In this work, we also tried bagging ensemble models, such as, Random forests and Gradient Boosted Decision Trees (GBDT).

Model	Error (in %)	Model	Error (in %)
Linear Reg	19.6	SVM Reg (Lin)	21.2
Ridge Reg	20.2	Decision trees	9
Lasso	18.9	SVM Reg (Poly)	16
Lars	20.34	Gaussian Processes	13
Elastic Net	21.68	Random forest	8.2
SGD	16.7	GBDT	8.1

Table 5: Comparison of model performances, showing average error of all experiments (of Figure 9) done on INTREPID.

The comparative performance of different regression models over all datasets for INTREPID is presented in Table 5. We skip details of each individual model because of space constraints; but after testing all models, we choose tree-based regression models since they resulted in the lowest test error across all datasets tested. Tree-based models include decision tree (for standalone classifiers) and random forests and GBDT (for ensemble models). Tree-based models are simple and intuitive to understand because the decision at each step (node of the tree) is based on a single attribute or feature (in our case system, parameter) of the dataset, which involves a quick look-up operation along the depth of the tree. In contrast, other machine learning algorithms build models in dual space or solve a complex optimization problem, which makes it difficult to judge the relative usefulness of specific dataset attributes. Moreover, unlike most machine learning models, tree-based models do not require

Independent (input) Variables					Tunable Parameters				Output Parameters	
total data block dimensions	data block dimensions per core	partici-pating processes	number of fields	system memory	number of aggre-gators	aggre-gation factor	number of files	whether using re-structuring	Mean throu-ghput	(with std dev)
GlobalV	LocalV	#Cores	V	Mem	AGP	AF	F	R	thput (mean)	thput (stddev)
1024×1024×512	64×64×64	2048	4	1	64	4	4	0	3260.491	272.334

Table 4: Example data point showing 9 attributes.

much parameter tuning. Indeed, tree-based models such as random forests and GBDT are a popular choice for model building and data analytics and have proved useful in related HPC applications [9]. In the discussions below, we present our results using tree-based models. For our experiments, we use Python-based standard regression packages from the open-source machine learning toolkit *scikit-learn* [31]. For all the experiments, we perform multiple runs and report the mean value.

Training Data. Our training data consists of performance figures for different parameter settings collected during the characterization study (see Section 4). We collect data from two phases: data shuffling and data I/O. We combine these datasets to construct training data for the entire two-phase I/O. Our system modeling and performance predictions are based on this combined dataset.

Attributes. We selected a wide range of attributes to improve the discriminative power of the model. To start with, we extracted application information in terms of global and local data resolution, number of variables (or fields), and file count. We also utilize system-level information, such as core count, aggregator count, aggregation factor, memory and whether restructuring was used or not. Overall we had a total of 9 attributes which are listed in Table 4 along with a brief description of each. Table 4 also presents an example line from our combined dataset. Attribute selection or attribute reduction procedures [9] to select a smaller set of useful attributes are usually beneficial for large attribute spaces. In our case, since the number of attributes was already small and moderately reasonable, we did not perform any attribute selection or reduction.

5.1 Validation results

We present results to demonstrate the accuracy of the model trained on the combined dataset. We have trained our model on training data obtained on small cores (1,024, 4,096 and 8,192) for the microbenchmarking application. In this section we first validate our model on test data also obtained from the same small core regime. Performance validation results are presented for micro-benchmarking and S3D applications, where aggregate throughput is predicted (given a set of parameters). Each run of S3D generates four variables; pressure, temperature, velocity (3 samples) and species (11 samples).

Performance Validation. Figure 9 present results of model validation on low core count regimes for microbenchmarking (2,048 cores) and S3D applications (4,096 cores). We predict the throughput for different values of the aggregator count. In all cases, the throughput prediction of the model is close to that of the original throughput values. Note that for

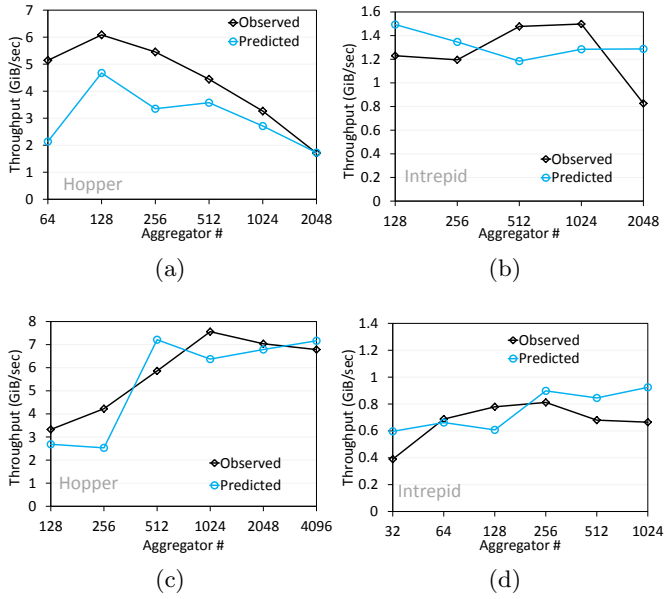


Figure 9: Validation results for microbenchmark (top row) and S3D (bottom row) for HOPPER (left panels) and INTREPID (right panels).

S3D the prediction performance is particularly good despite the fact that the model is trained on the microbenchmark data. Thus the trained model is sufficiently general and performs well across different target applications. In Figure 9, we see that the overall average percentage error for HOPPER is $\sim 32\%$ and for INTREPID is about 20%.

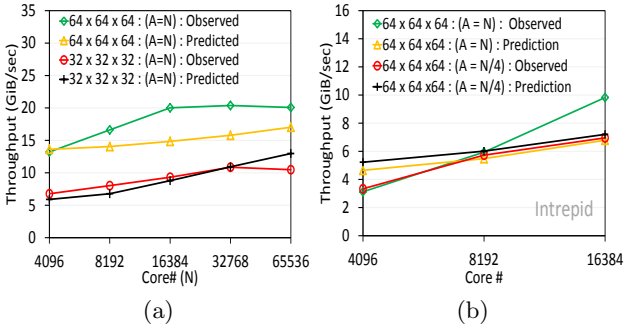

64 x 64 x 64 : (A=N) : Observed
64 x 64 x 64 : (A=N) : Predicted
32 x 32 x 32 : (A=N) : Observed
32 x 32 x 32 : (A=N) : Predicted
64 x 64 x 64 : (A = N) : Observed
64 x 64 x 64 : (A = N) : Prediction
64 x 64 x 64 : (A = N/4) : Observed
64 x 64 x 64 : (A = N/4) : Prediction
Throughput (GiB/sec)
Core# (N)
Core #
Intrepid
(a)
(b)

Figure 10: Prediction results at high core counts for S3D on (a) HOPPER and (b) INTREPID.

Parameter Prediction to Maximize Throughput. In this section, we predict parameters that maximize through-

put. To this end, we propose an adaptive modeling framework shown in Figure 11. We start with a model trained on the labeled datasets. Next we sample a set of candidate points $p_1, p_2, \ldots, p_n$ and predict the performance throughput of each point in the candidate set using model t^0. We select the point p^k that has the maximum performance throughput. Thereafter we set the machine to the parameter values obtained using the selected point p^k and run one time step of the simulation process. The output of the simulation process yields the true observed throughput p^k which we feed along with the point p^k, to the model (via the dotted loop) and retrain the model. This iterative process continues until we reach a predefined number of time steps or the predicted throughput does not change in subsequent iteration, indicating that the model has converged or is close to convergence.

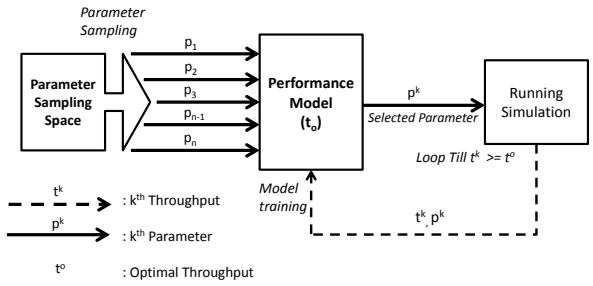


Figure 11: Block diagram for adaptive modeling.

We apply our model on Hopper test data at 4,096 cores and 64^3 data load. We observe that our model is able to identify the best A value 128. We note that this model is fully automatic and adapts its performance at each iteration to find parameter settings that improve the overall throughput performance. In future, we plan to test this model on larger core counts and use it to auto tune the system performance fully automatically and without any manual intervention.

5.2 Prediction for high core counts

In validation results, we test the performance of our prediction on the same regime from which training data has been collected. In this section, we take a leap of faith and study the performance of our prediction on high core count regimes (16K, 32K, and 64K). Note that since the model has not seen points in high core count regime, it would be useful to update them after prediction on each data point. This adaptive approach caters to *online* machine learning techniques (where we update the model after seeing each point) and is in contrast to the *offline* learning models used in the validation section.

For performance prediction, we consider different test cases in Hopper and Intrepid. For Hopper, we try different data loads 32^3 and 64^3 whereas for Intrepid we keep the load fixed 64^3 but vary the parameter settings. Figure 10 shows the modeling results for S3D on Hopper and Intrepid. Observe that for Hopper the prediction is better for a 32^3 data load than a 64^3 data load. On Intrepid, the prediction for $A = N/4$ is better than for $A = N$. In all cases, the predictions are reasonably close, showing the benefits of an adaptive model that gradually improves itself over multiple prediction time steps. For a 32^3 data load, the error on Hopper and both cases of Intrepid are less than 30%. For much larger 64^3 data load, however, the error increases drastically. As we can see, tree-based nonlinear models have been unable to capture this behavior of Hopper. We believe that the characteristics of Hopper at high data loads change significantly from that on low data loads and hence need additional investigations for characterization and modeling. One reason could be that the network phase in Hopper is much less well behaved than in Intrepid.

5.3 Discussion

In Figures 9 and 10, we see that modeling error is high in only a few cases (for example, for 64 aggregator core count in Figure 9(a)). In addition, we observe that the overall average percentage error was particularly high for Hopper (32% in Figure 9) but much less for Intrepid (20% in Figure 9). These results lead us to conclude that Intrepid is more well-behaved and simpler to model than is Hopper. Similarly, in characterization study plots Hopper exhibited higher variability in error than did Intrepid. One reason for the difference in behavior can be attributed to the fact that the network phase, which is a component of the combined throughput, is more well behaved on Intrepid and hence is easier to model and predict. This is primarily to do with how jobs are allocated by default on Blue Gene vs Cray machines. On Blue Gene the communication traffic is isolated, whereas on Cray it can interfere with the traffic of other jobs.

We note that our proposed model can be applicable to other parallel I/O library file formats, such as pnetCDF, parallel HDF5, and Fortran I/O. As for PIDX, in each case we need to identify discriminating attributes that we believe would be useful in training the model. For PIDX, we have used such information in the form of aggregator count, number of cores, number of files, and so forth. Similarly, for pnetCDF we can use number of files created [12] presented with subfiling approach for pnetCDF (where performance is demonstrated with respect to the number of files written), or any other parameter affecting performance. Once the informative attributes are identified, we can collect these feature values for different data points and create a training data set to train a model. Thus the underlying model remains unchanged, and we need only to vary the attributes or fields in order to make model applicable to different file formats.

In this work, we used the data from our initial characterization study to built a regression-based performance model. This performance model is beneficial for exploring parameter spaces that are difficult to cover in characterization studies. For instance, in Figure 10 we use the trained model to approximate performance up to 64K core counts. Our model also reinforces the performance pattern observed in the characterization study; for example, Figure 8(a) (of the characterization study) and Figure 9(a) (of the modeling) demonstrate similar trends of declining throughput with increasing aggregator counts.

6. RELATED WORK

Parallel scientific simulations often produce large volumes of data. In order to aid in structure and efficient access of these huge data sets, a variety of high level I/O libraries such as pnetCDF [25], Parallel HDF5 [1] and Parallel IDX [17, 19, 18] have been proposed. A significant amount of research

has been devoted to characterize and model the I/O and network behavior of High Performance Computing (HPC) systems. In this section, we discuss existing literature from the above two areas that we believe are relevant to our work.

Characterization study. In this section we look at previous done work related to I/O capability, performance, and scalability of leading HPC systems. A body of work has focussed on the characterization study of Jaguar, a Cray XT3/XT4 machine, at Oak Ridge National Laboratory. For example Yu et al. [41] studied the scalability for each level of the storage hierarchy (up to 8,192 processes). Fahey et al. [10] characterized I/O performance for file with constant sizes. Both leveraged insights from their studies to tune and optimize I/O performance for scientific applications but also presented challenges towards scaling I/O for larger core counts. More recently, Xie et al. [38] characterized bottlenecks of multi-stage I/O pipeline of Jaguar on the lustre filesystem. Their study used IOR for benchmarking and talked about the straggler phenomena where I/O bandwidth gets limited due to few slow storage targets (stragglers). Other existing I/O characterization works for Cray machines include [40] and [32].

Similar studies by Lang et al. [22] on the I/O characterization for IBM Blue Gene/P system have highlighted I/O challenges faced by IBM Blue Gene/P system Intrepid at the Argonne National Laboratory. The authors reports capacity of each I/O stage, studying individual components, building up to system-wide application I/O simulations. Another work [37] on the Blue Gene/P supercomputer presented topology-aware strategies for choosing aggregators for two-phase I/O. Focusing mainly on parallel file-systems, detailed study comparing performances of GPFS, Lustre and PVFS can be found at Oberg et al. [28].

Additionally, studies have also been carried out on characterizing the network and the inter-process communication patterns of HPC systems. For instance [26], [36] and [15] studies communication characteristics, performance tradeoffs along with comparisons among different interconnect technologies. Specifically Vetter et al. [36] uses MPI and hardware counters to examines the communication characteristics of several scientific applications covering both collective as well as point-to-point communication. Our characterization study (both I/O and network), on the other hand tries to understand system behaviors of two different architectures (Hopper and Intrepid) by exploring the entire parameter space (system as well as algorithmic), hence finding both concurring as well as contradicting patterns that effects performance on the two systems.

Performance Modeling. Performance modeling of large scale parallel systems has been addressed in Lublin et al. [27] where the authors modeled workloads to characterize supercomputers and claimed it to be better than trace-based modeling. Following [9], most of the existing work can be categorized as analytical models, heuristic models, and trial-and-error methods. Analytical models, proposed in [7, 42], are usually difficult to construct owing to the complexity of modern day multicore processors. An alternative line of work [23] developed heuristic performance model based on offline benchmarking. Heuristic models, a popular approach [5], are usually too closely tied to the underlying system and hence do not generalize beyond the target hardware and application. Current day machines are too complex and such heuristic performance modeling is tedious, difficult and error-prone in such scenarios. Trial-and-error methods [13], as the name suggests, try our numerous combinations but the number of combination become quickly unmanageable for current day hardware. Recently, numerous works focussed on using machine learning for auto tuning and performance studies. Machine learning models were used [9] to estimate performance degradation on multicore processors. A neural network based optimization framework that can dynamically select and control the number of aggregators was proposed in [20]. Genetic algorithm based auto tuning framework was proposed [3] for the parallel I/O library HDF5. Shan et al. [33] use IOR synthetic benchmark to parameterize I/O workload behavior and thereafter predict its performance on different HPC systems. Our, work also does prediction, but we instead have developed a performance model that predicts performance corresponding to varying parameter configurations without actually running the experiments. Another work [16] uses curve fitting techniques to derive equations for performance metric for understanding I/O behavior of HPC systems. In the fields of modeling storage, Randy et al. [24] discusses analytic performance models for disk arrays. Other relevant research related to performance modeling of I/O workloads include [35], [14], [2] and [11].

7. CONCLUSION

In this work, we have performed a characterization study and modeling of the parallel I/O library PIDX. We observed that owing to differences in ways job partitions are allocated, the two architectures exhibited different network scaling behaviors, for Hopper the network fails to scale at higher core counts. On the other hand, the Intrepid network is more stable, scalable and, less responsive to varying message sizes than Hopper. From our I/O characterization study, we observe that small details such as file creation time add substantial overhead. Hopper (Lustre) is more optimized to a unique file per process I/O approach than is Intrepid (GPFS), whose I/O is optimized for fewer shared files. We also observe that optimizing I/O at varying scale requires proper choice of aggregators which again varies according to the machine.

We use the datasets from our characterization study for training machine learning models. Our models show that throughput and parameters can be accurately predicted using regression analysis techniques. We also show that models trained on datasets from low numbers of cores perform reasonably well on high numbers of cores, albeit with some online adaptive updates. We note that the models we propose are independent of the characteristics of the target machine, the underlying file system, and the custom I/O library used and thus can be applied to other I/O application scenarios.

Acknowledgment

We thank Abhinav Bhatele, Todd Gamblin, Kate Isaacs, Aaditya Landge and Joshua Levine for help with installation and use of the Boxfish tool. This work was supported by the U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-

AC02-06CH11357 and an Argonne National Laboratory Director Fellowship. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

8. REFERENCES

[1] HDF5 home page. http://www.hdfgroup.org/HDF5/.

[2] K. Barker, K. Davis, and D. Kerbyson. Performance modeling in action: Performance prediction of a cray xt4 system during upgrade. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–8, 2009.

[3] B. Behzad, J. Huchette, H. Luu, R. Aydt, Q. Koziol, M. Prabhat, S. Byna, M. Chaarawi, and Y. Yao. Abstract: Auto-tuning of parallel io parameters for hdf5 applications. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1430–1430, 2012.

[4] C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*, volume 1. springer New York, 2006.

[5] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 557–558, New York, NY, USA, 2010. ACM.

[6] S. P. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[7] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[8] J. M. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News*, 21:31–38, December 1993.

[9] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, and J. Pei. A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 83:1–83:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[10] M. Fahey, J. Larkin, and J. Adams. I/o performance on a massively parallel cray XT3/XT4. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, 2008.

[11] D. Feng, Q. Zou, H. Jiang, and Y. Zhu. A novel model for synthesizing parallel i/o workloads in scientific applications. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 252–261, 2008.

[12] K. Gao, W.-K. Liao, A. Nisar, A. Choudhary, R. Ross, and R. Latham. Using subfiling to improve programming flexibility and performance of parallel shared-file I/O. In *International Conference on Parallel Processing (ICPP)*, pages 470–477, September 2009.

[13] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 22:1–22:14, New York, NY, USA, 2011. ACM.

[14] P. Hanuliak. Analytical method of performance prediction in parallel algorithms. *Open Cybernetics & Systemics Journal*, 6:38–47, 2012.

[15] W. Jiang, J. Liu, H.-W. Jin, D. Panda, W. Gropp, and R. Thakur. High performance mpi-2 one-sided communication over infiniband. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 531–538, 2004.

[16] S. Johnson Baylor, C. Benveniste, and L. Boelhouwer. A methodology for evaluating parallel i/o performance for massively parallel processors. In *27th Annual Simulation Symposium*, pages 31–40, 1994.

[17] S. Kumar, V. Pascucci, V. Vishwanath, P. Carns, R. Latham, T. Peterka, M. Papka, and R. Ross. Towards parallel access of multi-dimensional, multiresolution scientific data. In *Proceedings of the Petascale Data Storage Workshop (PDSW)*, November 2010.

[18] S. Kumar, V. Vishwanath, P. Carns, J. A. Levine, R. Latham, G. Scorzelli, H. Kolla, R. Grout, R. Ross, M. E. Papka, J. Chen, and V. Pascucci. Efficient data restructuring and aggregation for i/o acceleration in pidx. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 50:1–50:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[19] S. Kumar, V. Vishwanath, P. Carns, B. Summa, G. Scorzelli, V. Pascucci, R. Ross, J. Chen, H. Kolla, and R. Grout. PIDX: Efficient parallel I/O for multi-resolution multi-dimensional scientific datasets. In *IEEE International Conference on Cluster Computing*, 2011.

[20] S. Lakshminarasimhan, D. A. Boyuka, S. V. Pendse, X. Zou, J. Jenkins, V. Vishwanath, M. E. Papka, and N. F. Samatova. Scalable in situ scientific data encoding for analytical query processing. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, HPDC '13, pages 1–12, New York, NY, USA, 2013. ACM.

[21] A. Landge, J. Levine, A. Bhatele, K. Isaacs, T. Gamblin, M. Schulz, S. Langer, P.-T. Bremer, and V. Pascucci. Visualizing network traffic to understand the performance of massively parallel simulations. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2467–2476, 2012.

[22] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/o performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 40:1–40:12, New York,

NY, USA, 2009. ACM.

[23] B. Lee, R. Vuduc, J. Demmel, and K. Yelick. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In *International Conference on Parallel Processing (ICPP)*, pages 169–176 vol.1, 2004.

[24] E. K. Lee and R. H. Katz. An analytic performance model of disk arrays. In *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '93, pages 98–109, New York, NY, USA, 1993. ACM.

[25] J. Li, W.-K. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of SC2003: High Performance Networking and Computing*, Phoenix, AZ, November 2003. IEEE Computer Society Press.

[26] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. Panda. Performance comparison of mpi implementations over infiniband, myrinet and quadrics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 58–58, 2003.

[27] U. Lublin and D. G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *J. Parallel Distrib. Comput.*, 63(11):1105–1122, Nov. 2003.

[28] M. Oberg, H. M. Tufo, and M. Woitaszek. Exploration of parallel storage architectures for a blue gene/l on the teragrid. In *9th LCI International Conference on High-Performance Clustered Computing*, 2008.

[29] V. Pascucci and R. J. Frank. Global static indexing for real-time exploration of very large regular grids. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2001.

[30] V. Pascucci, G. Scorzelli, B. Summa, P.-T. Bremer, A. Gyulassy, C. Christensen, S. Philip, and S. Kumar. The ViSUS visualization framework. In E. W. Bethel, H. C. (LBNL), and C. H. (UofU), editors, *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*, Chapman and Hall/CRC Computational Science, chapter 19. Chapman and Hall/CRC, 2012.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python . *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[32] P. C. Roth. Characterizing the i/o behavior of scientific applications on the cray XT. In *International workshop on Petascale data storage (PDSW)*, PDSW '07, pages 50–55, New York, NY, USA, 2007. ACM.

[33] H. Shan, K. Antypas, and J. Shalf. Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, 2008.

[34] B. Summa, G. Scorzelli, M. Jiang, P.-T. Bremer, and V. Pascucci. Interactive editing of massive imagery made simple: Turning atlanta into atlantis. *ACM Trans. Graph.*, 30:7:1–7:13, April 2011.

[35] A. Uselton, M. Howison, N. Wright, D. Skinner, N. Keen, J. Shalf, K. Karavanic, and L. Oliker. Parallel i/o performance: From events to ensembles. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–11, 2010.

[36] J. S. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *J. Parallel Distrib. Comput.*, 63(9):853–865, Sept. 2003.

[37] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka. Topology-aware data movement and staging for i/o acceleration on blue gene/p supercomputing systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 19:1–19:11, New York, NY, USA, 2011. ACM.

[38] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki. Characterizing output bottlenecks in a supercomputer. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 8:1–8:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[39] C. S. Yoo, R. Sankaran, and J. H. Chen. Three-dimensional direct numerical simulation of a turbulent lifted hydrogen jet flame in heated coflow: flame stabilization and structure. *Journal of Fluid Mechanics*, pages 453–481, 2009.

[40] W. Yu, S. Oral, J. Vetter, and R. Barrett. Efficiency evaluation of cray XT parallel io stack. In *Cray User Group Meeting (CUG 2007)*, 2007.

[41] W. Yu, J. S. Vetter, and H. S. Oral. Performance characterization and optimization of parallel i/o on the cray XT. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–11. IEEE, 2008.

[42] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In J. C. Hoe and V. S. Adve, editors, *ASPLOS*, pages 129–142. ACM, 2010.